**Lab 8**

**Pipes**

**Edgar Cobos**

**Problem Description**

The goal of this lab is to introduce us students the concept of pipes and how they are used. This lab contains two processes, a parent and a child. The parent process places a message in a pipe and the child process reads the message from the same pipe. We will also write a second program that makes the two processes more independent by needed to specify a pipe’s name in order to read/write to and from it.

**Problems Encountered**

**Issue:** What happens when a process tries to read from a pipe before any data is written to it?

**Solution:** Doing some online manual research, I found that a read on a pipe will block if there is no data in the pipe. So it means that it will wait until something is written to the pipe.

**Issue:** What happens if f\_des[0] and f\_des[1], of a pipe, are interchanged?

**Solution:** The f\_des[0] location is designated solely for reading and the f\_des[1] location is designated solely for writing.

**C Code**

/\* Unnamed pipe pipe.c \*/

/\* Usage: pipe message\_to\_be\_written. Child write a message to parent \*/

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <unistd.h>

#include <string.h>

int main(int argc, char \*argv[]) {

int f\_des[2]; //use for the message queue

static char message[BUFSIZ];

char buffer[MAX\_CANON];

int i, k, n;

pid\_t childpid;

//make sure that the proper number of arguments are supplied

if (argc !=2) {

printf ("\n Usage: %s message\n", \*argv);

exit(1);

}

/\* generate pipe \*/

if (pipe(f\_des) == -1) {

//only execute if there is an issue creating the pipe

perror ("Pipe");

exit(2);

}

switch (fork()) {

case -1:

//only execute if there is an issue forking

perror ("Fork");

exit(3);

case 0: /\* In the child \*/

//delete the file descriptor for the read buffer

close(f\_des[0]);

//write the message to the buffer

if (write(f\_des[1], argv[1], strlen(argv[1])) != -1) {

printf ("Message sent by child: \*%s\*\n", argv[1]);

//flush the stdout stream

fflush(stdout);

}

else {

perror ("Read");

exit(4);

}

break;

default: /\* In the parent \*/

//delete the file descriptor for the write buffer

close(f\_des[1]);

//read a message from the buffer

if (read(f\_des[0], message, BUFSIZ) != -1){

printf ("Message read by parent: [%s]\n", message);

//flush the stdout stream

fflush(stdout);

}

else {

perror ("Write");

exit(5);

}

}

exit(0);

}

/\* Named pipe pipe1.c \*/

/\* Usage: pipe1 pipename. Child writes a message to parent \*/

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

#include <unistd.h>

#include <string.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <sys/wait.h>

#include <fcntl.h>

#include <errno.h>

# define BUFSIZE 256

int main(int argc, char \*argv[]) {

//declre the FIFO persmission

mode\_t fifo\_mode = S\_IRUSR | S\_IWUSR;

int fd, status, child;

char buf[BUFSIZE];

unsigned strsize;

//make sure that the proper number of arguments are supplied

if (argc !=2) {

printf ("\n Usage: %s pipename\n", argv[0]);

exit(1);

}

/\* generate a named pipe with r/w for user \*/

if ((mkfifo(argv[1],fifo\_mode) == -1) && (errno != EEXIST)) {

//only execute if there is and issue creating the pipe

perror ("Pipe");

exit(1);

}

if (( child = fork()) == -1) {

perror ("Fork");

exit(1);

}

else if (child == 0){

//child if goign to attempt opening the pipe

printf ("\nChild %ld is about to open FIFO %s\n", (long)getpid(), argv[1]);

if ((fd = open(argv[1], O\_WRONLY)) == -1) {

perror("Child cannot open FIFO");

exit(1);

}

/\* In the child \*/

// convert the long pid into a char\*

sprintf (buf, "This was written by child %ld\n", (long)getpid());

//string size needed for the write() function

strsize = strlen(buf) + 1;

//write the message to the queue

if (write(fd, buf, strsize) != strsize) {

printf("Child write to FIFO failed\n");

exit(1);

}

printf ("Child %ld is done\n", (long)getpid());

} else { /\* parent does a read \*/

//parent tries to read from the queue

printf ("Parent %ld is about to open FIFO %s\n", (long) getpid(), argv[1]);

if ((fd = open(argv[1], O\_RDONLY | O\_NONBLOCK)) == -1) {

//only execute if there is an issue opening the pipe

perror("Parent cannot open FIFO");

exit(1);

}

printf ("Parent is about to read\n");

//wait for the child process to exit

while ((wait(&status) == -1) && (errno == EINTR));

//parent tries to read from the queue (buffer)

if (read(fd, buf, BUFSIZE) <=0) {

perror("Parent read from FIFO failed\n");

exit(1);

}

printf ("Parent %ld received: %s\n", (long)getpid(), buf);

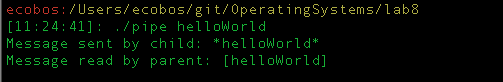
}

exit(0);

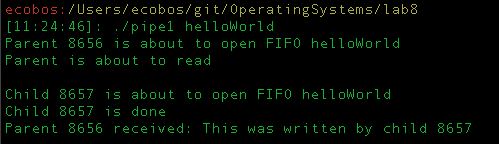
}

**Output**

This image shows pipe.c working properly. The child process sends a message to the parent process. The parent process waits for a message to be sent.



This image shows pipe1.c working properly. The name of the FIFO queue is “helloWorld”. The parent opens the FIFO pipe. The child write to the pipe and the parent reads from it.



**In *pipe*, does the order of f\_des[0] and f\_des[1] make any difference?**

The answer is yes and no. It does make a difference when you try to use the file descriptor located at f\_des[0] for writing. This is because the file descriptor in the index 0 location is designated for reading only. So if it’s used for writing then the program will throw a “Bad file descriptor” error message. The same applies for f\_des[1] when trying to read from that file descriptor, since f\_des[1] is designated for writing only. The only time when the order makes no different is the order or how data is read and written. For example, if a process tries to read from the pipe before a different process writes to it, then the read function blocks and waits for the write operation. So it will never be the case that a read operation read garbage due to a different process not writing yet.